Trends in Introductory Programming Courses in Australian Universities – Languages, Environments and Pedagogy

Raina Mason[1] **Graham Cooper**[1] **Michael de Raadt**[2]

[1] Southern Cross Business School
Southern Cross University
Hogbin Drive, Coffs Harbour, New South Wales 2450
raina.mason@scu.edu.au
graham.cooper@scu.edu.au

[2] Moodle Pty Ltd
1/224 Lord St, Perth, WA 6000
michaeld@moodle.com

Abstract

This paper reports the results of a study of 44 introductory programming courses in 28 Australian universities, conducted in the latter months of 2010. Results of this study are compared with two censuses previously conducted during 2001 and 2003, to identify trends in student numbers, programming language and environment/tool use and the reasons for choice of these, paradigms taught, instructor experience, text used and time spent on problem solving strategies in lectures and tutorials. Measures of mental effort experienced during the solution of novice programming problems were also examined.

Keywords: introductory programming, programming languages, programming environments, Australian university courses, mental effort measures, census, trends.

1 Introduction

Programming skills are an essential part of Information Technology (IT) and Computer Science (CS) courses. Programming is generally regarded to be both complex and difficult, and introductory programming courses can suffer from high attrition rates and low levels of competency (McCracken et al. 2001). There has been much debate in the academic community about what languages, environments and paradigms should be used for students' first exposure to programming (Bruce, 2005; Cooper, Dann, et al., 2003; Kelleher and Pausch, 2005).

In 2001 a census of introductory programming courses at Australian universities was conducted (de Raadt, Watson & Toleman, 2002), which reported on the languages and environments/tools being used, the reasons for the choice of language, student numbers and the paradigm being taught. The census covered 57 courses at 37 of the 39 Australian universities. The other two universities did not offer programming courses. The census was repeated in 2003 (de Raadt, Watson & Toleman, 2004) and expanded to include New Zealand universities. The 2003 census examined trends in languages and reasons for language choice, paradigms taught, tools and environments used, as well as new questions on texts employed, method of delivery to on-campus students, instructor experience and information about the teaching of problem solving strategies.

In the latter months of 2010 the census was to be repeated with all Australian universities. Participation was not as high as had been hoped, with a participation rate of 28 universities from the 39 that offered programming courses. A total of 44 out of 73 available programming courses were covered. While no longer a census, the study contains a large sample of the available first programming courses offered. The results of this 2010 study are reported in this paper, and have been compared to the results from the 2001 and 2003 censuses in order to identify longitudinal trends in language, tools and paradigms and to identify reasons for any such changes over the 10 year period. The basis for constructing interview questions and for conducting the study are described in the next section, followed by a discussion of the results and implications to teaching introductory programming.

2 Methodology

The previously collated list of participants from the 2001 and 2003 studies was used as a starting point for building a list of contacts for the current study. Information from university websites was also used to identify potential university programs and to collect contact numbers of administrative staff or academic staff responsible for those programs.

Once identified, each academic responsible for a particular introductory programming unit was sent an introductory email outlining the past two censuses. Participants were then contacted by phone within seven days to invite participation in the new census, and to arrange a convenient time for a phone interview of around 10 to 15 minutes duration.

All phone interviews were audio-recorded with the participant's permission. Notes on responses and comments were also entered manually into paper-based census forms as a backup to the recordings. Audio recordings were later transcribed and the data analysed.

It was found that the terminology used for a unit of study that is completed by students towards a degree varies between institutions, for example "subject", "course" or "unit" are used. The interviewer used the terminology particular to each institution when conducting an interview to reduce possible confusion from ambiguity. In the remainder of this paper, the results are reported using the description "course" for the basic unit of study (usually studied over the period of a semester or session, in conjunction with other units of study), to be consistent with the last two censuses.

2.1 Questions

Questions repeated from the 2001 and 2003 censuses probed language and paradigm choice, teaching duration, instructor experience, textbooks, problem solving strategies and development tools.

The orbit of enquiry of the 2003 census was expanded to include questions regarding the mental effort (Paas and van Merrienboer 1993) required to understand and learn aspects of programming using the language(s) used in each course.

On this basis, the following question was asked:

- "How difficult do you think this language is for students to learn?" (9 point Likert scale, 1 = extremely easy, 9 = extremely difficult)

If an interviewee indicated that he or she used an environment or tool beyond a simple editor and command line compiler, then the following questions were asked:

- "Why was this environment/tool chosen?"
- "Is this environment/tool used for an initial part of the first programming course only, or throughout the first programming course?"
- "Is the environment/tool used in any other courses in the degree? If so, how many? Is it used in a different way in subsequent courses?"
- "How difficult do *you* find the environment to use (on a scale of 1 to 9 where 1 is extremely easy, 9 is extremely difficult)"
- "On average how difficult do you believe *the students* find the environment to use?" (9 point Likert scale)

All participants were also asked questions about the mental effort expended when solving a novice programming problem, on each of three measures:

- understanding and processing the problem statement;
- navigating or using the environment, tools or language; and
- learning from the problem and reinforcing previous concepts.

Participants were asked to estimate the mental effort expended on each of these three measures by themselves, by an average student, and by a student in the bottom 10% of their course.

3 Results and discussion

The results of this study are reported below, with comparison to the previous two censuses. For more accurate comparison, only the data from Australian universities in the 2003 census has been used.

3.1 Universities and courses

The 2010 study covered 44 of the 73 programming courses offered by Australian universities. A total of 28 of the 39 universities offering programming courses participated in the study (see Table 1).

	2001	2003	2010
Universities	39	40	40
Universities teaching programming	37	39	39
Introductory Programming courses	57	71	(44) 73
Total students in study (approx.)	19900	16300	7743
Average students per course	349	229	176

Table 1: University/Course Summary

3.2 Participation rate

Three instructors declined to participate. Several instructors were employed casually and were not available on campus at any time apart from their face-to-face teaching commitments, two instructors had retired recently and were not available for comment, and some instructors (and administrative staff) were unavailable during the 3 month period of conducting interviews for this study. All but one of the participants agreed to the audio-recording of the interview. This participant agreed to be interviewed with hand-written notes being taken on paper-based census forms.

3.3 Number of courses

Although the total number of programming courses offered has changed little since the 2003 census, instructors stated that often business-based IT programming courses, computing science programming courses and engineering programming courses had been amalgamated into one course. The reasons offered by participants in several institutions were that management had strategically decided to merge different courses and associated student cohorts to gain efficiencies due to economies of scale. These actions had been in response to declining numbers of students. At the same time, programming courses have appeared in non-traditional programming areas such as visual arts, keeping the number of introductory programming courses relatively static.

3.4 Student numbers

The declining number of students studying programming is a serious problem. Average numbers of enrolments per course have halved, falling from 349 in 2001 to just 176 in 2010. This follows a general trend in declining student enrolments in all areas and levels of ICT education - vocational, undergraduate and postgraduate - with 17 436 total domestic ICT enrolments in 2001 falling to just 7 470 domestic students in 2008 (ACS, 2010).

This is despite strong growth prospects for employment in the IT industry, with 7 out of 11 ICT job designations in the "Job Prospects Matrix" (DEEWR, 2011) considered to have 'strong' to 'very strong' growth prospects for future employment. The widening gap between increasing demand and declining domestic student numbers may give rise to a significant short-fall

of suitably educated and skilled IT professionals in the near future.

3.5 Languages

3.5.1 Choice of language(s)

There were 20 different languages being taught by the academic staff interviewed, which is a greater diversity of languages than displayed in each of the last two studies. During 2001, only nine languages were recorded, and in 2003 this number had reduced to eight languages. In 2010, the number of languages taught over the duration of a course ranged from 1 to 6 (see Table 2), with the vast majority teaching just one language.

No. of languages	1	2	3 to 6
Courses	37	4	3

Table 2: number of languages in a course

Some languages were only used for a short time during a course, with another language being used for most of the course. For example, in one case Alice was used for the first two weeks, followed by Java. If only these 'primary' languages are counted, 12 languages were used by instructors. One of these languages is "MaSH", a language created as a staged subset approach to Java (Rock, 2011). For the purposes of this study, this language has been counted as a variant of Java. The number and percentage of courses using each primary language, as well as the weighted percentage by students, are shown in Table 3.

Language	Courses	%age	Weighted by students
Java	16	36.4%	38.4%
Python	6	13.6%	19.2%
C	5	11.4%	11.7%
C#	4	9.1%	8.0%
Visual Basic	4	9.1%	5.1%
C++	3	6.8%	4.8%
Processing	2	4.5%	5.2%
Alice	1	2.3%	0.9%
Fortran	1	2.3%	3.9%
Javascript	1	2.3%	1.5%
Matlab	1	2.3%	1.3%

Table 3: 2010 Languages

The set of top three languages has changed since the 2003 census. Java continues to hold the place of the most popular language, a result which is in accordance with a recent survey of programming courses in the USA (Davies et al. 2011). C++ and Visual Basic have decreased in popularity and fallen out of the top three. Python, under development since 1990, is the second most popular language in 2010, followed by C. According to the TIOBE programming language index (TIOBE Software 2011), industry use of Python had the greatest growth in popularity of any language in the years of 2007 and 2010.

The relatively new language C# is the fourth most popular language, seemingly replacing Visual Basic, which has dropped to just 5.1% of student capture. This apparent replacement is not surprising, as both Visual Basic and C# are Microsoft products and C# is considered to be more modern and popular than Visual Basic (TIOBE Software 2011).

A comparison of the language use across the 2001, 2003 and 2010 studies is provided in descending order by percentage of courses (Table 4) and by student numbers (Table 5).

Language	2001	2003	2010
Java	40.4%	40.8%	36.4%
Python	0.0%	0.0%	13.6%
C	7.0%	12.7%	11.4%
C#	0.0%	0.0%	9.1%
VB	24.6%	26.8%	9.1%
C++	14.0%	11.3%	7.0%
Processing	0.0%	0.0%	4.5%
Fortran	0.0%	1.4%	2.3%
Javascript	0.0%	0.0%	2.3%
Matlab	0.0%	1.4%	2.3%
Alice	0.0%	0.0%	2.3%
Haskell	5.3%	4.2%	0.0%
Eiffel	3.5%	1.4%	0.0%
Delphi	1.8%	0.0%	0.0%
Ada	1.8%	0.0%	0.0%
jBase	1.8%	0.0%	0.0%

Table 4: language comparison by courses

Language	2001	2003	2010
Java	43.9%	44.4%	39.0%
Python	0.0%	0.0%	19.5%
C	5.5%	10.6%	11.9%
C#	0.0%	0.0%	8.2%
Processing	0.0%	0.0%	5.3%
VB	18.9%	16.4%	5.2%
C++	15.2%	18.7%	4.9%
Fortran	0.0%	0.7%	3.9%
Javascript	0.0%	0.0%	1.5%
Matlab	0.0%	1.0%	1.3%
Alice	0.0%	0.0%	0.9%
Haskell	8.8%	6.0%	0.0%
Eiffel	3.3%	2.1%	0.0%
Delphi	2.0%	0.0%	0.0%
Ada	1.7%	0.0%	0.0%
jBase	0.8%	0.0%	0.0%

Table 5: Language comparison by students

Figures 1 and 2 chart the changes in popularity of the top 4 languages in each of these three studies (Java, VB, C++

and Haskell in 2001, Java, VB, C++ and C in 2003, and Java, Python, C and C# in 2010), by percentage of courses offering the language, and then weighted by students.

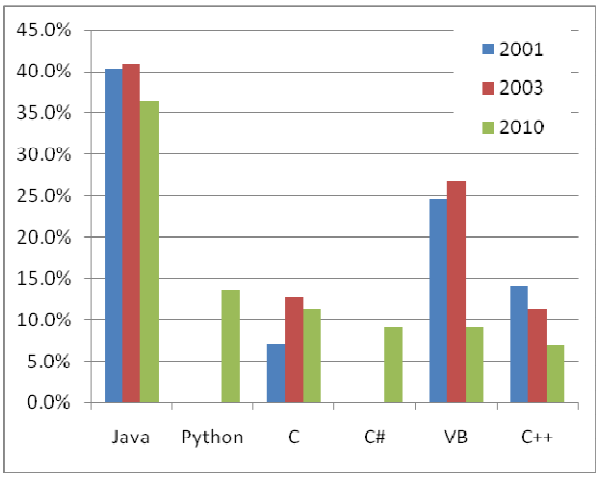

Figure 1: Top 4 languages of each study (by %age of courses)

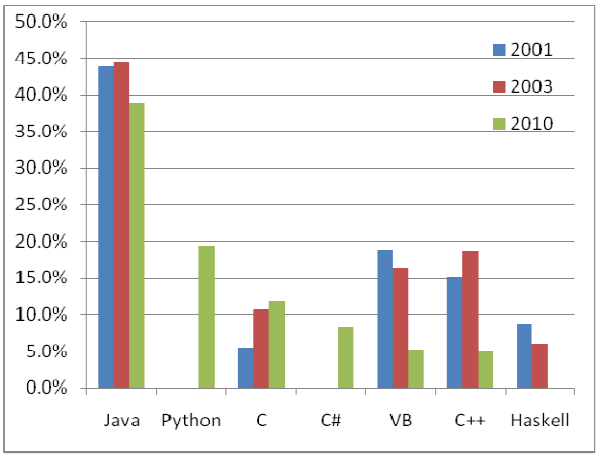

Figure 2: Top 4 languages of each study (weighted by student numbers)

3.5.2 Reason for choice of language

Instructors were asked in the 2001 census and the 2010 study about the reasons for their choice of language. More than one reason could be offered.

In 2001 the most commonly provided reason for choosing a language was industry relevance and/or marketability to students (with 56.1% of participants identifying this as a reason). The second most commonly provided reason for choosing a language was "pedagogical benefits", with one third (33.3%) of the instructors presenting this as a reason for language choice. The results of the 2010 study presented a substantial shift in the frequency of these reasons being given for language choice, with industry relevance and marketability declining (to 48.8%) and pedagogical benefit rising (to 53.5%).

The 2010 survey also identified the emergence of several new reasons. Instructors mentioned "the availability of a community and online help", "extendability and libraries available", "platform independence" (as opposed to "limitations of OS/machines" as in the 2001 census), "ease of installation", and "interpreted language", with no need to compile. Some of these reasons reflect the rise of the open source community over the time period, as well as perhaps a greater choice of operating systems by students. Table 6 shows the change in percentages of instructors' reasons for language choice between 2001 and 2010, ordered by frequency in 2001.

Reason	2001	2010
Used in industry / marketable	56.1%	48.8%
Pedagogical benefits of language	33.3%	53.5%
Structure of degree/dept politics	26.3%	32.6%
OO language	26.3%	16.3%
GUI interface	10.5%	7.0%
Availability/Cost to students	8.8%	4.7%
Easy to find appropriate texts	3.5%	2.3%
OS/Machine limitations of dept	1.8%	4.7%
Online community and help available	0%	9.3%
Platform independence	0%	9.3%
Extensions/Libraries available	0%	7.0%
Interpreted language	0%	4.7%
Ease of installation	0%	2.3%

Table 6: Reasons for language choice

Figure 3 shows the comparative frequencies of reasons given in the 2001 census with the 2010 results, ordered by the 2010 results.

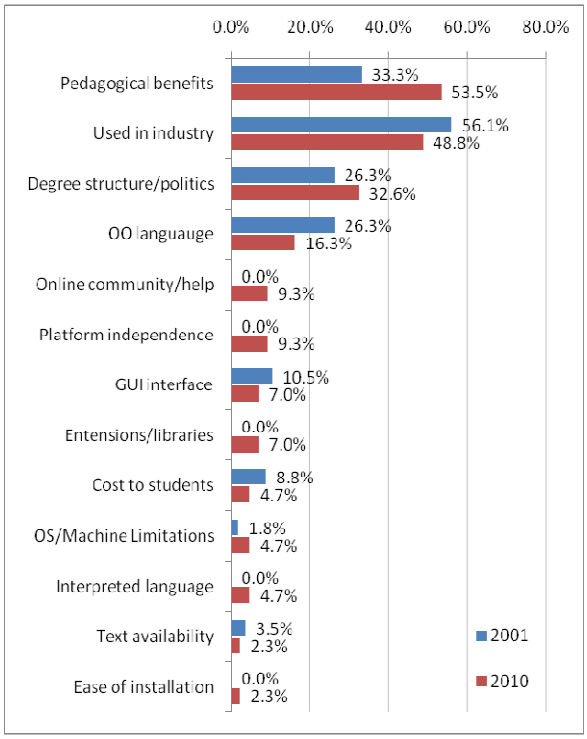

Figure 3: Trends in reasons for language choice

This figure clearly shows that although "industry relevance and marketability to students" is still seen as important, the reasons for choosing a language have

shifted to be more strongly inclusive of pedagogical factors associated with both ease of learning and availability of support.

The intersection of industry relevance and pedagogy can be seen in the reasons given by those instructors who chose Python as their teaching language. As previously noted, Python is one of the more popular languages according to the TIOBE Programming Community Index (TIOBE Software 2011), which indicates the popularity of programming languages in industry (numbers of skilled engineers world-wide and third-party vendors) and in training courses. However *all* of the instructors who chose Python commented that the reason they chose the language was because it was perceived as easier for students to learn. Only one instructor gave 'industry relevance' as a secondary reason for the choice of Python, and this was due to its association with the Google Apps engine. The comment was also made that "we have to cater for prep [sic] students and keep IT students happy, so we have to find a balance between these".

The reason "structure of degree/department politics" was the only other reason given in 2001 that increased in frequency in the 2010 study. As previously mentioned, in several cases two or more introductory programming courses in various disciplines such as engineering, business and computer science had been merged into one course. The language that was chosen for this one course became either a legacy language from one of the previous courses, or a language chosen to try to cater to a broader profile of students pushed into a narrower stream of programming teaching. Several instructors expressed frustration at the need to cater to a range of students with differing backgrounds, experience and capabilities. Typical comments included "We see students with a range of skills - from no experience to some with some computing in high school" and "IT students are not the same as CS students".

3.6 Paradigm taught

In common with the 2001 and 2003 censuses, the 2010 study showed that most instructors choose to teach using a procedural paradigm. Some instructors - 8 from the 44 interviewed - also reported that they taught primarily procedurally but introduced some object-oriented concepts. For example, they may mention objects or use the other terminology of object-oriented programming to prepare students for further courses that covered OO programming. Some used objects but "in a procedural way". For the purposes of comparison and consistency with previous studies, these have been designated under the procedural paradigm.

Some instructors described how they used either a mix of paradigms - for example they started with procedural and then used the last 6 weeks to cover OO concepts - or suggested that they taught more than one language and taught each language in different ways. The number of courses teaching in each paradigm is given in Table 7.

Longitudinal trends in paradigms taught are shown in Figure 4. The downwards trend in the numbers of instructors teaching objects-first is clearly shown and the use of the functional paradigm has nearly disappeared.

Paradigm	Courses	%age
Procedural	24	54.5%
Object-Oriented	11	25.0%
Mixture	8	18.2%
Functional	1	2.3%

Table 7: Paradigms taught

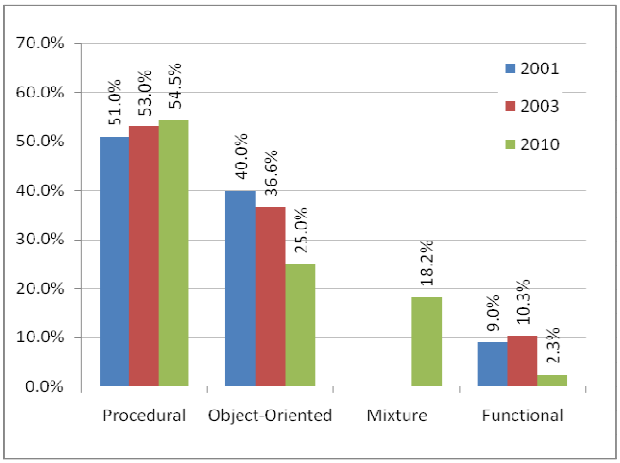

Figure 4: Trends in paradigms taught

3.7 On-campus hours

Instructors were asked about the time that on-campus students spent in lectures, tutorials and practicals each week. Although the average hours spent in each did not differ greatly from 2003 (see Table 8), several institutions stated that they either have no lectures at all, instead presenting in a more interactive 'workshop' format, or that all lectures were available only online via online classroom software and/or audio or video recordings. Most instructors commented that all course materials were available online and that often students did not attend classes, instead choosing to download materials from home, even when the course was not offered by distance education.

Several interviewees commented on the success of the 'workshop' approach: "Workshops are a much more successful way of teaching programming. Give them small bits and then have them do it straight away."

	Lecture	Tutorial	Practical	TOTAL
2003	2.2	0.6	1.8	4.6
2010	2.1	1.1	1.2	4.4

Table 8: Hours in class on-campus

3.8 Instructor Experience

Instructors were asked how many years they had taught introductory programming. The average amount of experience has risen since 2010 (see Table 9). However it should be noted that casual teachers were often not available for interviews, and this may have skewed the results.

	Minimum	Average	Std Dev	Maximum
2003	0.5	8.6	7.2	30
2010	2	12.3	7.3	30

Table 9: Instructor experience in years

3.9 Texts used

Of the 44 courses in this current study, 11 used no text at all, and many instructors commented that they encouraged students to use online resources or to find their own text. Most of the remainder used language, or environment-specific texts, with no real commonalities. Four more generic texts - "Connecting with Computer Science" (Anderson, Hilton and Ferro 2010), "Simple Program Design" (Robertson 2000), "Programming Language Concepts" (Sebesta 2007) and "A simple and generic introduction to OO Algorithm Design" (Robey undated) - were used, each in only one course.

3.10 Problem Solving Strategies

Instructors were asked how much time they dedicated during class time to teaching and discussing problem solving strategies within the context of programming.

The average percentage time given to problem solving, along with standard deviations, have remained stable from 2003 to 2010, and are presented in Table 10. The percentage of times given to problem solving in both lectures and tutorials remain unchanged in this period.

	Lecture		Tutorial	
	Average	Std Dev	Average	Std Dev
2003	29%	22%	46%	36%
2010	29%	26%	44%	33%

Table 10: percentage of class time dedicated to problem solving

It is worth noting that while the means between years are very close for both lectures and tutorials, the standard deviations on these measures are relatively large, indicating substantial variations between different courses in how much time they dedicate explicitly to problem solving.

Some participants indicated that 'problem solving' had been moved into a separate course and so did not deal with these strategies in the programming course. In contrast, others stated that problem solving was implicit in everything they did in the lectures and tutorials, although they did not explicitly teach problem solving strategies.

Although the average percentage of time given to focusing upon problem solving has remained static from 2003 to 2010, there may be differences in how problem solving is embedded within curriculum structures, classroom delivery and learning activities between these two dates. This information, however, lies beyond the bounds of the current paper.

3.11 IDEs and tools

3.11.1 Choice of IDE/tools

Some languages (including Alice and Processing) require the use of a specific environment. Instructors who did have a choice of environments or tools chose a wide variety or none at all. Microsoft Visual Studio was the most popular IDE. The use of the teaching environment BlueJ continued to increase, from 4% in 2001 to 17.5% in 2010. Within the 'other' category, note should be made of the "IDLE" IDE (for use with Python), at 12.5% and Eclipse, used by 7.5% of instructors.

The largest change has been the movement away from using text editors and command line compilers only - from approximately 45% of these instructors using no IDE/tool in 2001 and 2003, to just 20% in 2010. Figure 5 shows the trends in environment use over the course of the three studies.

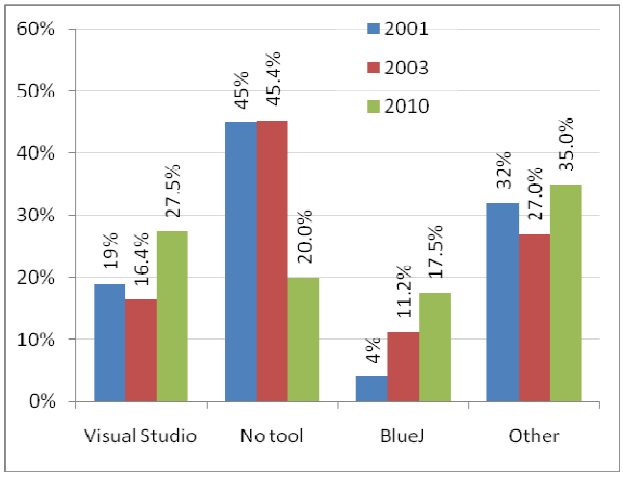

Figure 5: Trends in environment use

Two instructors reported using a specialised learning environment (such as Alice or BlueJ) for an initial part of the course, followed by a more industry-standard IDE such as Visual Studio or Eclipse. Others used the learning environment throughout the first programming course.

3.11.2 Reasons for choice of environment

At the time of the 2003 census, most instructors were choosing to *not* use an IDE, if they were not forced to do so by their choice of language. Anecdotally this was due to the perceived overhead of instructing students on that tool, hence instructors preferred to use a simple editor and command line compiler. During the 2010 study, instructors were directly asked about the reason or reasons that they had chosen a particular IDE or tool. The top 10 reasons for choosing an environment/tool (excluding those where the language is part of the environment, such as Alice and Processing) is given in Table 10.

Other reasons included "associated text is very good", "cross-platform", "plugins available", "ease of installation" and "GUI". Some instructors offered reasons as to why they did not choose to use an IDE, even though this was not explicitly asked as part of the study. These included "students need to become familiar with the command line" and "we don't have time for that".

Reason	Count	% courses
Pedagogical reasons	15	41.7%
Packaged with language	14	38.9%
Cost to students	13	36.1%
OS limitations of dept.	7	19.4%
Uncomplicated/Ease of use	6	16.7%
Industry relevance	4	11.1%
Supports OO paradigm	4	11.1%
Visual cues/Visual debugger	4	11.1%
Student motivation	3	8.3%
Open source	3	8.3%

Table 10: Top 10 reasons for environment choice

The most frequent reason given for choice of environment (provided by 41.7% of instructors) was pedagogically-based. This is consistent with the reasons provided by participants for choice of language, where pedagogical reasons was also identified as the most frequent reason for choice (of language).

Associated comments offered by instructors emphasised that some environments assisted learning by allowing students to concentrate on the concepts of programming, rather than the specifics and nuances of a language:

- "can think about objects instead of thinking about the language itself";
- "[an] introduction without the stress of having to worry about syntax rules. It reinforces what happens when you use a loop so the frog jumps once or it just keeps jumping. You have to understand the concept, so its concept reinforcement.";
- "We found in the past that students were having trouble with understanding what all the features of the main method are, and some of the initial concepts we had to abstract away – we had to say 'treat this as magic, you have to do it'";
- "...mostly to encourage students who have no programming background, and strengthens the concepts of loops, and iterations and all those things".

Some instructors stressed that the environment chosen was simple whilst still including tools to reduce the complexity of compiling and building:

- "it's really a smarter text editor with buttons for compiling and building. Nothing like Visual Studio or Eclipse. …so we can concentrate on language syntax";
- "To reduce the amount that students had to learn in order to get to the heart of programming";
- "just its simplicity";
- "simple, not confusing".

Others pointed at an IDE's perceived intrinsic superiority to simple editors/command line compilers, due to the inclusion of helpful tools:

- "why get them to travel by horse when they can travel by car?";
- "the tools that it provides for students are wonderful compared to using just a basic text editor - they offer nothing".

There has been an apparent shift from viewing the use of an IDE as an additional overhead, to seeing it as reducing the amount a student has to learn, or as a tool to help the student to learn. Whether this viewpoint is correct is debatable, and may be a function of what is being taught (central concepts to programming in general, or a language specific syntax), the student's ability and experience, and aspects of the language or environment itself.

3.12 Mental effort

3.12.1 Novice programming, mental effort and cognitive load

Mental Effort refers to the level of conscious focus of attention one has to give to a task, whether it is cognitive, physical, or a bit of both (Paas et al. 2003).

International educational and scientific computing body 'The Association for Computing Machinery (ACM)' (2008) suggests that learning the three generic concepts of sequence, iteration and selection is an integral part of all first programming courses. A novice programmer will need to acquire this knowledge base as schemas (Chase & Simon 1973) and automate them, whereby they can be applied with relatively low levels of conscious attention (Cooper & Sweller 1987).

Novice programming problems that require the student to learn and use any of these three concepts can be said to have three sources of mental effort:

- understanding the problem and what is required, and deciding on the best structures to use to solve the problem;
- navigating and using the environment, tools and language, in an attempt to solve the problem; and
- learning how to best use these structures so they can be used in further, more difficult problems.

These three aspects equate to three identified sources of cognitive load on working memory while learning: intrinsic, extraneous and germane (Sweller, van Merrienboer & Paas 1998).

Intrinsic cognitive load refers to the innate relative difficulty of a body of to-be-learned information. It is effectively set, defined by the content.

Extraneous cognitive load refers to the load generated by the format of instructional materials and/or to the performance of learning activities. Some formats and/or activities hinder learning by loading the learner with unnecessary information and/or tasks. This source of cognitive load is variable, determined by the learning materials.

Germane cognitive load refers to load devoted to the processing, construction and automation of schemas - knowledge structures in long-term memory. This is not simply a measure of motivation, but refers to the dedicated commitment of cognitive resources to the successful process of cognitive acquisition of new to-be-learnt information.

The general strategy sought in many instructional settings is to reduce extraneous load, and to direct the

subsequently released cognitive resources towards the germane efforts associated with schema acquisition and automation. The term 'mental effort', defined above, is used in the current study as a means of evaluating the *cognitive load* (Sweller 1998) associated with various aspects of learning programming.

3.12.2 Measures of mental effort

Instructors were asked about the three sources of mental effort expended whilst solving a novice programming problem:

- understanding and processing the problem statement,
- navigating or using the environment, tools or language, and
- learning from the problem and reinforcing previous concepts.

Instructors were asked to rate their own levels of mental effort on each of these three factors using a 9 point Likert scale, where 1 = "no mental effort" and 9 = "extreme mental effort". Instructors were subsequently asked to rate their expectations regarding these three factors of mental effort for an average student in their introductory programming course, and then again, for a student in the "bottom 10%" of the course performance.

Table 11 shows the mean, median and mode for each of these cognitive load areas, for instructors, the average student and students in the 'bottom 10%'.

		instructor	average student	bottom 10% student
intrinsic	mode	2	6	9
	median	2	6	8.5
	mean	2.8	6.0	7.8
	std dev	1.8	1.6	2.1
extraneous	mode	2	5	9
	median	2	5	8
	mean	2.4	4.9	7.7
	std dev	1.1	1.5	1.3
germane	mode	2	7	9
	median	2	5.5	8.5
	mean	3.2	5.6	7.6
	std dev	2.1	1.9	2.3

Table 11: Levels of mental effort

Note that there were some participants who did not quantify a response for an aspect of this series of questions, particularly for the 'bottom 10%'. Instead they offered comments and discussion. These are removed from this first series of analyses, and are beyond the scope of the current paper, but will be explored in a further paper.

3.12.3 Comparisons of mental effort

For each of these three areas of cognition (understanding the problem statement, using the environment, and reinforcing previous concepts) a series of Wilcoxon Signed Rank tests were performed, firstly comparing the self-rating of the instructor to that anticipated to be experienced by an 'average student', and then comparing the anticipated level to be experienced by an average student to one who is in the "bottom 10% of students".

Table 12 shows the results, using Wilcoxen Signed-rank test (one-tailed) for all measures:

In summary, these results indicate that for *each* of these three sources of cognitive load, the instructors in the introductory programming courses rated their own levels of required mental effort to be low, and that they expected that average students would need to exert higher levels of mental effort than themselves - 'above average'.

Additionally, for *each* of these three sources of cognitive load, the participants rated the anticipated mental effort to be experienced by a student in 'the bottom 10%' to be higher again, compared to an average student, in the rating of high to extreme mental effort.

Instructor -> average student (greater mental effort)	W	Ns/r	z	p	n
Understanding and processing the problem statement	811	43	4.39	<0.0001	43
Navigating/using the environment, tools or language	838	41	5.43	<0.0001	43
Learning from the problem/ reinforcing previous concepts	647	40	4.34	<0.0001	42
Average -> bottom 10% student (greater mental effort)	W	Ns/r	z	p	n
Understanding and processing the problem statement	207	24	2.95	0.0016	25
Navigating/using the environment, tools or language	276	23	4.19	<0.0001	25
Learning from the problem/ reinforcing previous concepts	144	20	2.68	0.0037	21

Table 12: Wilcoxen Signed-rank test

Average students need to "work harder" with their cognitive resources than the lecturer, and for less able students this is further exacerbated. This indicates that it is unlikely that these students will learn effectively, no matter how much effort they may put in.

Further research is needed to explore the reasons why some students struggle and the ways in which various environments and/or languages may aid or hinder learning these generic concepts in programming.

4 Further discussion

The shrinking number of students enrolled in introductory programming courses continues to be a concern. Average numbers of students per course have approximately halved in the 10 years since the first census was conducted. This is echoed by the ACS (2010) figures which show a drop in domestic ICT enrolments of 57% over the time period 2001 to 2008.

However ICT enrolments in Australian universities stayed relatively constant over the time period 2006 - 2008 (the latest figures available), primarily as a result of an influx of international student enrolments (ACS 2010). At the time of the latest figures in 2008, over 60% of the ICT enrolments at Australian universities were

international students, and the number of domestic ICT students were still decreasing (Table 13).

	Domestic		International		Total
2006	8198	44.8%	10087	55.2%	18285
2007	7839	43.0%	10384	57.0%	18223
2008	7470	38.6%	11896	61.4%	19366

Table 13: ICT enrolments 2006 - 2008

Languages taught in Australian universities continue to be dominated by Java, however there is a much wider diversity of languages than at the time of the last census. Instructors were also asked whether they had plans to change the first language, and although only four from 44 answered affirmatively, four others communicated that a change was being considered.

Languages that are seen as particularly beneficial for learning purposes (rather than for industry use) are becoming more popular, such as Python, Alice and Processing. This is supported by the stated reasons for choice of language or languages - even though "industry relevance" is still important, pedagogical factors were the reason for most instructors' language choice.

Of interest were the few courses that introduced novice users to three or more languages. The choice of several languages was either made because students needed to know these languages for further units, or as an attempt to show similarities in constructs and approaches in programming problem solving, despite differences in syntax and development environment. The mental effort results reported in this study suggest that for novices and in particular, for less able students, this latter approach is problematic at best, resulting in excess cognitive load and ineffective learning.

Online communities and resources have become more important. There were 9.3% of instructors gave "online community/help" as a factor in their choice of language, and a quarter of instructors set no text, many commenting that they encouraged students to use online resources.

Another change that has appeared since the last running of census is that students are anecdotally using a wider range of operating systems, and this has influenced instructors' choice of languages and environments. "Platform independence" was given as a reason for language choice by 9.3% of instructors, and was also mentioned as a factor in choice of IDE choice.

The focus on the object-oriented paradigm and the objects-first approach to learning programming appears to have reduced since the 2003 census. Those teaching objects-first dropped from 36.6% of courses in 2003 to just 25% in 2010. In addition, the reason "object-oriented language" for language choice was given by just 16.3% of instructors in 2010 compared to 26.3% in 2001. Interestingly, the use of BlueJ, an environment which supports the teaching of objects-first using Java, increased from 4% of courses to 17.5% over the same period.

The shifts towards greater emphasis upon pedagogy and pedagogical reasons for choices of languages and environments invites further exploration into some of the cognitive overheads experienced by students, particularly less able students in the cohort. The mental effort measures and differences reported in this paper indicate that instructors are aware of the higher levels of mental effort experienced by average students over that expended by themselves solving the same problem, and that less able students are in many cases experiencing 'extreme' cognitive load while trying to solve novice programming problems.

Most instructors indicate that the 'bottom 10%' of students may represent more than one student profile - for example, those who are trying but failing to succeed, those who don't try, and those who are absent. These comments are explored further in a separate paper.

5 Further work

The numbers of students enrolled in introductory programs, as well as domestic ICT enrolments as a whole, continue to trend downwards, and more investigation is required to determine why this is happening.

Several participants in this study suggested that their courses had been formed by the amalgamation of more than one course that was originally part of computer science, information technology, engineering or business programs. An exploration of the current percentages of each cohort of students in these new courses, and their relative success would be of interest.

The authors of this paper are also exploring the implications of cognitive load, and its three primary sources, being intrinsic to the complexity of the content, extraneously related to the instructional materials and environments, which for programming includes the role of languages and environment, and germanely through the conscious, deliberate, focus of attention to the acquisition and automation of associated schemas.

Ironically, the students who are most in need of acquiring and automating schemas, those in the lower end of ability in a student cohort, are those that are least capable of doing so, due to excessive levels of cognitive load. Of primary interest is the potential for selection of language and environment to hold the potential of lowering a source of extraneous cognitive load, to thus free cognitive resources for application to germane usage, with the intent of facilitating learning.

6 Acknowledgements

The authors would like to thank the participants in this study for their involvement.

7 References

Anderson, G., Hilton, R. & Ferro, D. (2010): Connecting with Computer Science. 2nd Ed. Cengage Learning.

Association for Computing Machinery (2008): Computing Curricula - Information Technology 2008. Curriculum Guidelines for Undergraduate Degree Programs in Information Technology. http://www.acm.org//education/curricula/IT2008 Curriculum.pdf. Accessed 29 Aug 2011.

Australian Computer Society (ACS) (2010): Australian ICT Statistical Compendium 2010, http://www.acs.org.au/2010compendium Accessed 18 Mar 2011.

Bruce, K.B. (2005): Controversy on how to teach CS 1: A discussion on the SIGCSE-members mailing list. *Inroads - The SIGCSE Bulletin* **37:** 111-117.

Chase, W.G. & Simon, H.A. (1973): Perception in chess. *Cognitive Psychology* **4** (1): 55–81.

Cooper, G., & Sweller, J. (1987): Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of Educational Psychology* **79** (4): 347–362.

Cooper, S., Dann, W. and Pausch, R. (2003): Teaching objects-first in introductory computer science. *Proc. ACM 34th SIGCSE technical symposium on Computer science education,* New York NY, USA, 191-195, ACM Press.

Davies, S., Polack-Wahl, J.A. & Anewalt, K., 2011. A snapshot of current practices in teaching the introductory programming sequence. *SIGCSE '11 Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*. ACM Press, pp. 625 - 630.

De Raadt, M., Watson, R. and Toleman, M. (2002): Language trends in introductory programming courses. *Proc. Informing Science and IT Education Conference,* Cork, Ireland, Cohen, E. and Boyd, E. (Eds). InformingScience.org

De Raadt, M., Watson, R. and Toleman, M. (2004): Introductory programming: what's happening today and will there be any students to teach tomorrow? *Proceedings of the sixth conference on Australasian Computing Education.* Dunedin, New Zealand, **30:** , Australian Computing Society, Inc.

DEEWR (2011): Australian Jobs 2011. http://www.deewr.gov.au/Employment/ResearchStatistics/Pages/AustralianJobs.aspx. Accessed 22 Aug 2011.

Kelleher, C. and Pausch, R. (2005): Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR),* **37**(2):83-137. ACM Press.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagam, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I. and Wilusz, T. (2001): A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin.* **33**(4):125-180.

Paas, F.G.W.C & van Merrienboer, J.J.G. (1993): The efficiency of instructional conditions: An approach to combine mental-effort and performance measures" *Human Factors* **35**(4): 737-743.

Paas, F, Tuovinen, J. E., Tabbers, H. & Van Gerven, P.W.M. (2003): Cognitive Load Measurement as a Means to Advance Cognitive Load Theory. *Educational Psychologist.* **38**(1):63-71. Lawrence Erlbaum Associates, Inc.

Robey, M. (undated): A simple and generic introduction to OO algorithm design. Self-published. http://www.computing.edu.au/~mike/ST151Book.pdf Accessed 24 Aug 2011.

Rock, A. (2011): MaSH (Making Stuff Happen). http://www.ict.griffith.edu.au/arock/MaSH/ Accessed 23 Aug 2011.

Robertson, L.A. (2000): Simple Program Design. Nelson Australia.

Sebesta, R. (2007): Programming Language Concepts. Addison Wesley.

Sweller, J. (1988): Cognitive load during problem solving: Effects on learning. *Cognitive Science* **12** (2): 257–285.

Sweller, J., Van Merriënboer, J., & Paas, F. (1998). Cognitive architecture and instructional design. *Educational Psychology Review* **10** (3): 251–296.

TIOBE Software (2011): TIOBE Programming Community Index - Long Term Trends. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Accessed 22 Aug 2011.